1. Tempo disponibile 120 minuti.
2. Non è possibile consultare appunti, slide, libri, persone, siti web, ecc.
3. Scrivere in modo leggibile, su ogni foglio, nome, cognome e numero di matricola.
4. Le soluzioni agli esercizi che richiedono di progettare un algoritmo devono:
   - spiegare a parole l'algoritmo (se utile, anche con l'aiuto di esempi o disegni),
   - fornire e commentare lo pseudo-codice (indicando il significato delle variabili),
   - calcolare la complessità (con tutti i passaggi matematici necessari),
   - se l'esercizio ammette più soluzioni, a soluzioni computazionalmente più efficienti e/o concettualmente più semplici sono assegnati punteggi maggiori.

1. Calcolare la complessità $T(n)$ del seguente algoritmo `mystery` assumendo implementazione della struttura UnionFind tramite quickUnion con euristica sul rango:

```
algoritmo mystery(n: Int) --> Int
  uf = new UnionFind()
  for i = 1 to n
    uf.makeSet(i)
  endfor
  u = 1; v = n
  while (u <= n)
    uf.union(uf.find(u), uf.find(v))
    u = u+2; v = v-1
  endwhile
  return uf.find(1)
```

Soluzione Le operazioni su strutture quickUnion con euristica sul rango hanno costo costante, per quanto riguarda `makeSet` e `union`, e logaritmico, per quanto riguarda `find`. L'algoritmo `mystery` esegue un primo `for` che esegue n cicli, ognuno di costo costante, ed un secondo `while` che esegue $n/2$ cicli, ognuno di costo $O(2 \times \log n) = O(\log n)$. Le restanti operazioni hanno costo costante. Il costo complessivo risulta essere $O(n + n/2 \times \log n) = O(n \log n)$.

2. Si consideri un albero binario T contenente valori interi e non ripetuti. Scrivere un algoritmo che dato in input l'albero T e due valori interi a e b, con $a < b$, restituisce *true* se e solo se T è un albero binario di ricerca le cui chiavi sono tutte comprese nell'intervallo $[a, b]$ (estremi inclusi).

Soluzione Per risolvere l'esercizio si può usare una visita ricorsiva dell'albero verificando, per ogni nodo, che il valore sia compreso nell'intervallo considerato e che i sottoalberi sinistro e destro siano a loro volta ABR. Per fare questo bisogna far attenzione a "restringere" opportunamente l'intervallo quando si invoca ricorsivamente il metodo sui nodi figlio.

```
isABR(node t, int a, int b ) --> boolean
       if ( t == null ) return true;
       else
              return ( (a ≤ t.value) &&
                       (t.value ≤ b) &&
                       isABR(t.left, a, t.value) &&
                       isABR(t.right, t.value, b) );
       endif
```

La complessità computazione nel caso pessimo è $\Theta(n)$ dove n è il numero di nodi. Si verifica quando T è un ABR ed è quindi necessario attraversare interamente l'albero. Il caso ottimo è $O(1)$ e si verifica se la radice non è compresa nell'intervallo [a,b].

3. Si consideri il seguente balletto medievale ballato da n maschi e n femmine. Inizialmente tutti i ballerini si collocano lungo una linea retta a distanza di un passo l'uno dall'altro. In una fase iniziale, le femmine, una alla volta, si muovono per raggiungere un maschio con cui formeranno una coppia per il resto del balletto. Le femmine si muovono a tempo, e ad ogni battuta della musica, la femmina che si sta muovendo fa un passo spostandosi a proprio piacimento verso destra oppure sinistra. Una volta raggiunto il compagno con cui formerà la coppia, alla battuta successiva un'altra femmina inizierà i propri spostamenti. Data una disposizione iniziale dei ballerini, bisogna capire quanto tempo sarà necessario per completare la formazione delle coppie. Il tempo di formazione di una coppia coincide con il numero complessivo di passi che devono effettuare le femmine per completare la formazione delle coppie. Progettare quindi un algoritmo che dato un vettore di booleani $B[1..2n]$ che indica le posizioni iniziali dei ballerini ($B[i] = true$ indica che in posizione i è presente una femmina, altrimenti se $B[i] = false$ in posizione i è presente un maschio), restituisce un intero che rappresenta la durata minima possibile per la formazione di tutte le n coppie (ovvero, il numero minimo complessivo di passi che le femmine devono fare per formare le coppie).

Soluzione Il problema può essere risolto tramite un approccio greedy, abbinando l'i-esima femmina con l'i-esimo maschio. Per calcolare il tempo per completare la formazione delle coppie, si sommano i passi che le femmine devono fare per raggiungere il proprio maschio abbinato. Gli abbinamenti vengono calcolati inserendo gli indici delle posizioni delle ballerine, e dei ballerini, in due distinte code FIFO. Successivamente si estraggono gli indici abbinati e si calcola la distanza considerando il valore assoluto della differenza degli indici.

```
algoritmo Balletto(B: Bool[1..2n]) --> Int
  Queue maschi = new Queue()
  Queue femmine = new Queue()
  for j = 1..2n
    if (B[j]) then femmine.enqueue(j)
    else maschi.enqueue(j)
  endfor
  Int tot = 0
  for j = 1..n
    tot = tot + abs(femmine.dequeue() - maschi.dequeue())
  endfor
  return tot
```

4. Progettare un algoritmo che, dato un grafo orientato $G = (V, E)$ ed un vertice $v \in V$, restituisce il numero di vertici di un ciclo di lunghezza minima a cui v appartiene (restituisce ∞ se v non appartiene ad alcun ciclo).

Soluzione È possibile usare una visita in ampiezza partendo dal vertice v. Infatti, in questo modo si visitano gli archi in ordine di distanza non decrescente a partire da v. Appena si visita un nodo (a distanza superiore a 1) adiacente a v, si può concludere che la lunghezza del ciclo minimo coincide con la distanza di tale nodo più 1. Se si termina la visita senza trovare alcun ciclo, tale ciclo non esiste e si può restituire ∞.

```
algoritmo CicloMinimo(Grafo G=(V,E), Vertice v) --> Number
    Queue q = new Queue()
    for each x in V
        x.mark = false; x.dist = INFINITY
    endfor
    v.mark = true
    v.dist = 0
    q.enqueue(v)
    while (not q.isEmpty())
        u = q.dequeue()
        if (u.dist>1 && v adiacente a u) return u.dist + 1
```

Cont.

```
        for each w adiacente a u
            if (not w.mark)
                w.mark = true
                w.dist = u.dist + 1
                q.enqueue(w)
            endif
        endfor
    endwhile
    return INFINITY
```

Il costo computazionale dell'algoritmo coincide con quello della visita in ampiezza, ovvero $O(m + n)$ con m numero di archi ed n numero di vertici del grafo. Si noti che a differenza della visita in ampiezza, nel caso in cui esista il ciclo, tale algoritmo può terminare prima di aver visitato l'intero grafo. Ma nel caso pessimo, che si verifica in assenza di tale ciclo, si rende necessario visitare l'intero grafo.

The End.